一、内存泄漏

应该被回收的对象，却无法被回收。

产生原因：一个对象应该被回收，但是有另一个正在使用的对象存在该对象的引用，导致无法被GC回收。

造成的影响：

1. 应用卡顿，泄漏的内存影响了GC的内存分配，过多的内存泄漏会影响应用的执行效率。
2. OOM，大量的内存泄漏会导致OOM.

AndroidStudio自带的内存分析工具分析：Dump Java Heap:

二、内存溢出

当应用的heap资源超过了Dalvik虚拟机分配的内存就会内存溢出.

### GC工作原理

1. 垃圾回收机制

垃圾收集算法必须做两件事情：首先检测出垃圾对象，其次必须回收垃圾对象所使用的堆空间并还给程序。

从程序的主要运行对象(如静态对象/寄存器/栈上指向的堆内存对象等)开始检查引用链，当遍历一遍后得到上述这些无法回收的对象和他们所引用的对象链，组成无法回收的对象集合，而其他孤立对象（集）就作为垃圾回收。

区分活动对象和垃圾的两个基本方法是引用计数和跟踪。

引用计数收集器、跟踪收集器

引用计数收集器：

如果增加一个该对象的引用，那么该对象的引用计数加1，如果减少就减1，引用计数为0的对象应该被回收。

优点：执行的速度比较快，可以在程序运行中执行。

缺点：无法检测出循环引用的情况。导致不能被回收。

跟踪收集器（标记并清除）：

追踪从根节点开始的对象引用图。在追踪过程中对遇到的对象打标机，追踪结束后，未被标机的对象应该被收集。

针对推碎块，有两种方法：压缩、拷贝。

压缩：把活动的对象越过空闲区滑动到堆的一端，堆的另一端出现一个大的连续空闲区。所有被移动的对象的引用也被更新，指向新的位置。

拷贝：把所有活动对象移动到一个新的区域。一般被称为停止-拷贝，堆被分为两个区域，任何时候只能使用一个区域。区域用尽时，程序中止，堆被遍历，从一个区域拷贝到另一个区域。

优点：对象可以在根对象开始遍历的过程中随着发现而拷贝，不再有标记和清除的区分。

缺点：需要两倍大小的堆内存，但是任何时候只能使用其中一半。每一次收集时，所有的活动对象都会被拷贝。而有的对象生命周期很长，所以会被来回拷贝，消耗大量时间。

分代是Java垃圾收集的一大亮点，根据对象的生命周期长短，把堆分为3个代：Young，Old和Permanent，根据不同代的特点采用不同的收集算法。

年轻代：

采取拷贝收集的方法。GC较频繁。少量，多次。当年轻代的内存使用达到一定的阀值时，Minor Collection就被触发。

年老代：

年轻代的对象如果能够挺过数次收集，就会进入老人区。老人区使用标记整理算法。因为老人区的对象都没那么容易死的，采用复制算法就要反复的复制对象，很不合算，只好采用标记清理算法。

持久代：

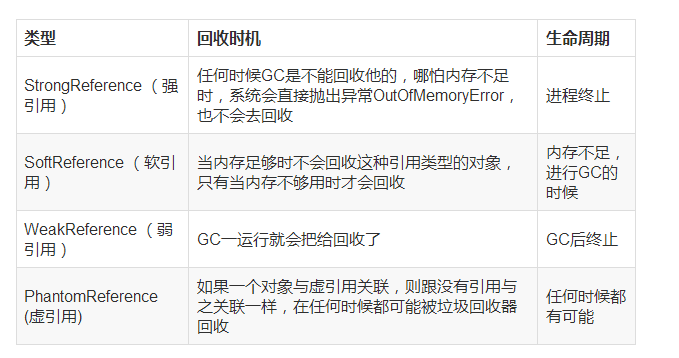
装载Class信息等基础数据。般情况下，持久代是不会进行GC的。

当年老代或者永久代的内存使用达到一定阀值时，一次基于所有代的GC就触发了，其特定是涉及范围广（量大），耗费的时间相对较长（较慢），但是频率比较低（次数少），称之为Major Collection(Full Collection)。通常，首先使用针对年轻代的GC算法进行年轻代的GC，然后使用针对年老代的GC算法对年老代和永久代进行GC。

自适应收集器：根据不同的情况选择不同的方法。

Android虚拟机的垃圾回收采用的是根搜索算法。GC会从根节点（GC Roots）开始对heap进行遍历。到最后，部分没有直接或者间接引用到GC Roots的就是需要回收的垃圾，会被GC回收掉。

1. GC过程与对象的引用类型关系



1. JVM内存分配
2. 静态的

静态存储区.

，内存在编译时就已经分配好，程序运行期间一直存在，存储静态数据、全局变量。

1. 栈式的

存储定义的变量，对象的引用。程序运行结束自动被释放。栈内存空间有限，大小由操作系统决定，Windows下栈为2M，连续的，不会产生碎片，运行效率高。

1. 堆式的

存储动态创建的对象。GC会根据内存的使用情况，对堆内存里的垃圾内存进行回收。堆内存是一块不连续的内存区域，如果频繁地new/remove会造成大量的内存碎片，GC频繁的回收，导致内存抖动，这也会消耗我们应用的性能。

1. ANR

Activity 5 s, BroadCast 10s。

1. 序列化

序列化：使得对象可以传输和存储。Java对象变成字节流的形式传出去，有的时候我们想要从一个字节流中恢复一个Java对象。

序列化方式：实现Serializable接口 和 Parcelable

Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。

Parcelable是Android为我们提供的序列化的接口,Parcelable相对于Serializable的使用相对复杂一些,但Parcelable的效率相对Serializable也高很多。需要自己控制变量是否被序列化。

1. Service

启动Service 两种方法：通过Activity 启动Service startService();bindService();

startService:启动后一直运行，调用自身的stopSelf方法或者调用stopService()方法停止Service.service 生命周期：onCreate，onStartCommand,onDestory.

bindService: 生命周期：onCreate，onBind,onDestory.通过unbindService方法解绑。

多次启动Service不会创建新的实例。

Service和Activity通信：

通过IBinder,Activity 绑定Service时，会调用OnBind方法，该方法会返回一个IBInder对象，自定义继承一个IBinder对象，作为返回值。在Activity中实现ServiceConnection接口类，ServiceConnection接口中有两个方法：onServiceConnected 和 onServiceDisconnected，这两个方法分别会在Activity与Service建立关联和解除关联的时候调用。onServiceConnected方法中可以得到Service中返回的IBinder对象。然后就可以调用IBinder中的方法了。

创建的Service是运行在主线程中的，IntentService 是继承自 Service 并处理异步请求的一个类，在 IntentService 内有一个工作线程来处理耗时操作。当任务执行完后，IntentService 会自动停止，不需要我们去手动结束。因为自己创建Service需要手动结束。重写onHandleIntent方法。

1. Anroid框架



从下往上依次为

* 内核层：Linux 内核和各类硬件设备的驱动，这里需要注意的是，Binder IPC 驱动也是在这一层实现，比较特殊
* 硬件抽象层：封装「内核层」硬件驱动，提供可供「系统服务层」调用的统一硬件接口
* 系统服务层：提供核心服务，并且提供可供「应用程序框架层」调用的接口
* Binder IPC 层：作为「系统服务层」与「应用程序框架层」的 IPC 桥梁，互相传递接口调用的数据，实现跨进层的通讯
* 应用程序框架层：这一层可以理解为 Android SDK，提供四大组件，View 绘制体系等平时开发中用到的基础部件

1. JVM

JVM的主要任务是装载class文件，并且执行其中的字节码。JVM中包含一个类装载器，可以从程序和API中装载class文件。Java API中只有程序执行时需要的类才会被装载。字节码有执行引擎来执行。

Java中有两种方法：Java 方法和本地方法。Java方法是由Java语言编写，编译成字节码。本地方法是由其他语言编写的，编译成和处理器相关的及其代码。本地方法保存在动态链接库中。通过本地方法，java程序可以直接访问底层操作系统的资源。

一个Java程序可以使用两种类装载器：启动类装载器和用户定义的类装载器。

启动类装载器是JVM 实现的一部分。也被称为原始类装载器、系统类装载器、默认类装载器、

用户定义的类装载器能够使用用户自定义的方式来装载类，比如从网络上下载class文件。用户定义的类装载器不是JVM实现的部分，它能够用Java编写，被编译为class文件，能够被虚拟机装载，实际上只是运行中的Java应用程序可执行代码的一部分。

被装载的类默认情况下只能看到被同一个类装载器装载的别的类。

Java class文件是可以运行在任何支持Java虚拟机的硬件平台和操作系统上的二进制文件。

Java API 是运行库的集合，它提供一套访问主机系统资源的标准方法。，Java API调用本地方法。

Java虚拟机包含三种含义：抽象规范、一稿具体的实现、一稿运行中的虚拟机实例。

一稿运行时的Java虚拟机的任务是运行一个Java程序。如果在同一台计算机上同时运行三个Java程序，将得到三个Java虚拟机实例，每个Java程序都运行与自己的Java虚拟机实例中。

Java虚拟机实例通过调用某个初始类的main（）方法来运行一稿Java程序。

Java程序初始类中的main方法，将作为该程序初始线程的起点，其他任何线程都是由这个初始线程启动的。

在Java虚拟机中有两种线程：守护线程和非守护线程。守护线程通常是由虚拟机自己使用的，比如执行垃圾收集任务的线程。但是，Java程序也可以把它创建的任何线程标记为守护线程。Java初始线程（main）是非守护线程。只要还有任何非守护线程在运行，那么这个Java程序也在运行。当所有的非守护线程都终止时，虚拟机实例将自动退出。

某些运行时数据区是由查询中所有线程共享的，有一些只能由一个线程拥有。每个Java虚拟机实例都有一个方法区和一个堆，它们是由该虚拟机实例中的所有线程共享的，当虚拟机装载一个class文件时，它会从这个class文件包含的二进制数据中解析类型信息。然后把这些类型信息放到方法区中。当程序运行时，虚拟机会把所有该进程在运行时创建的对象都放在堆中。

当每一个新线程被创建时，她都将得到自己的PC寄存器（程序计数器）以及一个Java栈。如果线程正在执行的是一个Java方法，那么PC寄存器的值将总是指示下一条将被执行的指令。

**运行时数据区**：

**方法区**：虚拟机实例所有线程共享，存储类型信息（全限定名，类类型or接口类型，类型的修饰符（public，abstract，final），字段名，字段类型，方法名，方法返回类型，参数类型、静态变量等等），。

**堆**：虚拟机实例所有线程共享，存储动态创建的对象。

**Java栈**：线程独享，存储该线程中Java方法调用的状态，包括局部变量，被调用时传进来的参数，返回值，以及运算的中间结果。

**PC寄存器**：线程独享，指示下一条将被执行的指令。

**本地方法栈**：存储本地方法的调用状态。

Java数据类型：基本类型、引用类型。

虽然Java虚拟机也把boolean看做基本类型，但是编译为字节码时，会用int或byte来表示boolean。在Java虚拟机中，false是由整数0来表示的，所有非0整数都表示true。涉及Boolean值的操作则会使用int。Boolean数组是当byte数组来访问。

Java虚拟机有一个只在内部使用的基本类型：returnAddress，这个基本类型被用来实现Java程序中的finally字句。

用户自定义的类装载器必须派生自ClassLoader类。

对于每一个被装载的类型，Java虚拟机都会为它创建一个Class类的实例来代表该类型。

1. Activity生命周期

onCreate,onStart,onResume,onPause,onStop,onDestroy.

onStart: Activity已经显示出来，但是我们看不到，相当于是在后台。

OnResume: Activity 已经可见了，OnResume 的时候Activity才显示到前台。

onPause: onPause 必须先执行完，新Activity的onResume才能执行。

如果新Activity采用了透明主题，那么当前Activity不会回调onStop。

OnCreate 对应 onDestroy; onStart 对应onStop, onPause 对应onResume.

默认情况下，手机横竖屏切换会导致Activity被消耗并且重新创建。

通过配置configChanges=orientation属性可以防止系统重新创建Activity。

Activity在异常情况下终止，系统会调用onSaveInstanceState来保存当前Activity的状态。调用时机在onStop之前。与onPause没有时序关系。当Activity被重新创建后，系统会调用onRestoreInstanceState，并把消耗时onSaveInstanceState方法所保存的Bundle对象作为参数传递给onRestoreInstanceState和onCreate方法。onRestoreInstanceState调用时机在onStart之后。

每一个View都有onSaveInstanceState和onRetorInstanceState方法。

Activity的启动模式：

Standard：标准模式。

SingleTop：栈顶复用，如果Activity位于栈顶，则不会被重新创建，否则会重新创建。同时会调用onNewIntent方法。

SingTask：栈内复用，如果所需的任务栈存在，如果Activity栈内存在，则将Activity调到栈顶。如果不存在则创建。如果所需的任务栈不存在，则创建任务栈，再创建Activity 压入栈。同时会执行onNewIntent方法。

SingleInstance：单实例模式，加强的singleTask。为启动的Activity创建一个新的任务栈，后续不会创建新的Activity。

IntentFilter中的过滤信息有：action，category，data。

1. IPC 机制

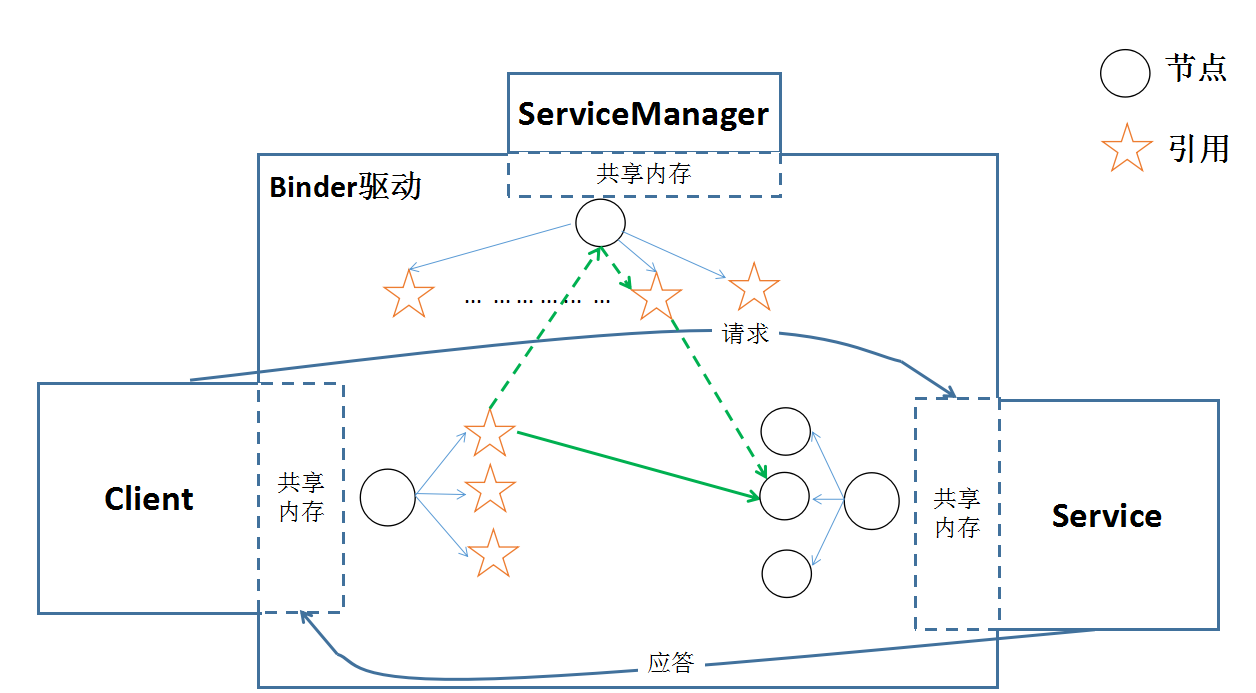
开启多进程：指定android:process属性。

Android为每一个应用分配了一个独立的虚拟机，为每一个进程都分配一个独立的虚拟机，不同的虚拟机有不同的地址空间，这就导致在不同的虚拟机中访问同一个对象会产生多个副本。不同的进程的组件会有独立的虚拟机，Application，以及内存空间。

实现跨进程的方式：Intent传递数据，共享文件和sharepreference，基于Binder的Messenger和AID，Socket。

Binder机制：

客户端持有远程进程的某个对象引用，然后调用引用类中的函数，远程进程的函数就执行了。



服务端跨进程的类都要继承Binder类。我们所持有的Binder引用(即服务端的类引用)并不是实际真实的远程Binder对象，我们的引用在Binder驱动里还要做一次映射。也就是说，设备驱动根据我们的引用对象找到对应的远程进程。客户端要调用远程对象函数时，只需把数据写入到Parcel，在调用所持有的Binder引用的transact()函数，transact函数执行过程中会把参数、标识符（标记远程对象及其函数）等数据放入到Client的共享内存，Binder驱动从Client的共享内存中读取数据，根据这些数据找到对应的远程进程的共享内存，把数据拷贝到远程进程的共享内存中，并通知远程进程执行onTransact()函数，这个函数也是属于Binder类。远程进程Binder对象执行完成后，将得到的写入自己的共享内存中，Binder驱动再将远程进程的共享内存数据拷贝到客户端的共享内存，并唤醒客户端线程。